Backpropagation is the core of how a neural network learns during the training process. It involves two key steps:

1. **Forward Propagation**: Data is passed through the network, and predictions are made based on current weights.
2. **Backward Propagation**: Errors are propagated backward to update the weights in order to minimize the error.

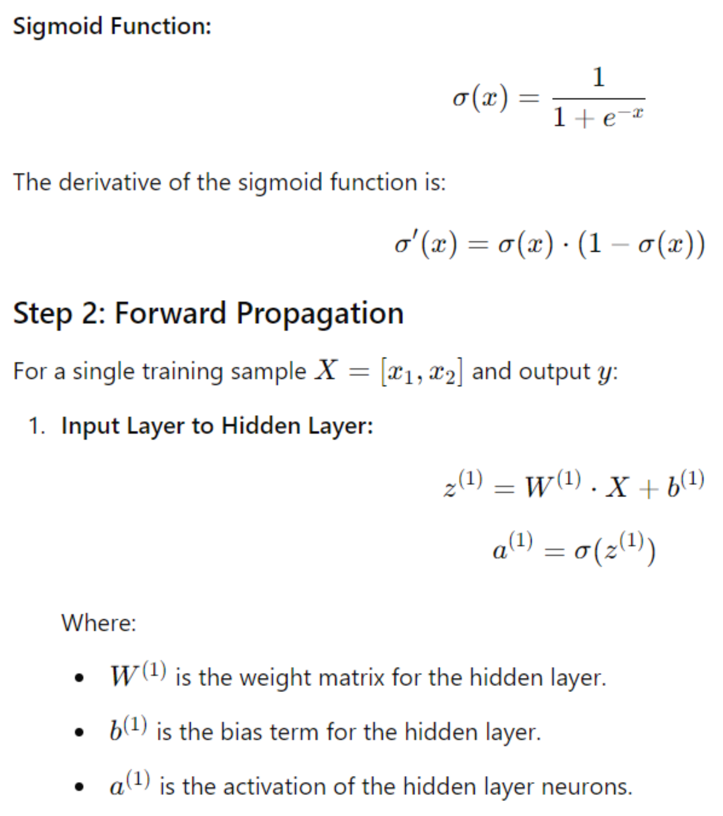
Let's break down backpropagation and weight updates step by step, both mathematically and through code, using a simple neural network.

Step 1: Simple Neural Network Architecture

We'll consider a neural network with:

- **2 input neurons**
- **1 hidden layer** with 2 neurons
- **1 output neuron**

Each layer will have weights that we will update using the backpropagation algorithm. We'll use a simple activation function called the **sigmoid function**.



Sigmoid Function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The derivative of the sigmoid function is:

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$$

Step 2: Forward Propagation

For a single training sample $X = [x_1, x_2]$ and output y:

1. **Input Layer to Hidden Layer:**

$$z^{(1)} = W^{(1)} \cdot X + b^{(1)}$$

$$a^{(1)} = \sigma(z^{(1)})$$

Where:

- $W^{(1)}$ is the weight matrix for the hidden layer.
- $b^{(1)}$ is the bias term for the hidden layer.
- $a^{(1)}$ is the activation of the hidden layer neurons.

2. **Hidden Layer to Output Layer:**

$$z^{(2)} = W^{(2)} \cdot a^{(1)} + b^{(2)}$$

$$a^{(2)} = \sigma(z^{(2)})$$

Where:

- $W^{(2)}$ is the weight matrix for the output layer.
- $b^{(2)}$ is the bias term for the output layer.
- $a^{(2)}$ is the prediction (output).

Step 3: Loss Function

We will use **Mean Squared Error (MSE)** as the loss function to calculate how far the predicted value is from the actual value.

$$L = \frac{1}{2}(a^{(2)} - y)^2$$

Step 4: Backpropagation (Error Propagation)

1. **Output Layer Error:** The derivative of the loss function with respect to the output $a^{(2)}$:

$$\delta^{(2)} = \frac{\partial L}{\partial a^{(2)}} = (a^{(2)} - y) \cdot \sigma'(z^{(2)})$$

2. **Hidden Layer Error:** The error propagated backward from the output layer to the hidden layer:

$$\delta^{(1)} = \left(W^{(2)}\right)^T \delta^{(2)} \cdot \sigma'(z^{(1)})$$

Step 5: Weight Updates

1. **Update the Weights of the Output Layer:**

$$W^{(2)} = W^{(2)} - \alpha \cdot \delta^{(2)} \cdot (a^{(1)})^T$$

2. **Update the Weights of the Hidden Layer:**

$$W^{(1)} = W^{(1)} - \alpha \cdot \delta^{(1)} \cdot X^T$$

Where:

- α is the learning rate.
- $\delta^{(1)}$ and $\delta^{(2)}$ are the errors for the hidden and output layers, respectively.

Code Example: Simple Neural Network from Scratch

```python
import numpy as np

# Sigmoid Activation Function and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# Neural Network class
class SimpleNeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        # Initialize weights and biases randomly
        self.W1 = np.random.randn(hidden_size, input_size)  # Weights for hidden layer
        self.b1 = np.random.randn(hidden_size, 1)           # Bias for hidden layer
        self.W2 = np.random.randn(output_size, hidden_size) # Weights for output layer
        self.b2 = np.random.randn(output_size, 1)           # Bias for output layer

    def forward(self, X):
        # Forward pass
        self.z1 = np.dot(self.W1, X) + self.b1
        self.a1 = sigmoid(self.z1)        # Activation for hidden layer

        self.z2 = np.dot(self.W2, self.a1) + self.b2
        self.a2 = sigmoid(self.z2)        # Activation (output) for final layer

        return self.a2

    def backward(self, X, y, learning_rate):
        # Calculate error (output layer)
        delta2 = (self.a2 - y) * sigmoid_derivative(self.a2)

        # Calculate error (hidden layer)
        delta1 = np.dot(self.W2.T, delta2) * sigmoid_derivative(self.a1)

        # Update output layer weights and biases
        self.W2 -= learning_rate * np.dot(delta2, self.a1.T)
        self.b2 -= learning_rate * np.sum(delta2, axis=1, keepdims=True)

        # Update hidden layer weights and biases
        self.W1 -= learning_rate * np.dot(delta1, X.T)
        self.b1 -= learning_rate * np.sum(delta1, axis=1, keepdims=True)
    def train(self, X, y, epochs, learning_rate):
        for epoch in range(epochs):
            # Forward pass
            output = self.forward(X)

            # Backpropagation
            self.backward(X, y, learning_rate)

            if epoch % 1000 == 0:
                loss = np.mean((y - output) ** 2)  # MSE Loss
                print(f"Epoch {epoch}, Loss: {loss}")

# Sample data (XOR problem)
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]).T  # 2 inputs
y = np.array([[0], [1], [1], [0]]).T              # 1 output

# Instantiate the neural network
nn = SimpleNeuralNetwork(input_size=2, hidden_size=2, output_size=1)

# Train the network
nn.train(X, y, epochs=10000, learning_rate=0.1)

# Test the network
print("Predictions after training:")
print(nn.forward(X))
```

Explanation of Code:

1. **Initialization**: The weights and biases are initialized randomly.
   - $W1$ and $W2$ are the weights for the hidden and output layers.
   - $b1$ and $b2$ are the biases.
2. **Forward Propagation**: The `forward` method computes the activations of both the hidden and output layers.
3. **Backward Propagation**: The `backward` method computes the gradients (errors) and updates the weights using gradient descent.
4. **Training**: The `train` method runs for a specified number of epochs, performing forward and backward propagation on each epoch.

Running the Code:

This neural network is trained on the XOR problem. After training, it will learn to approximate the XOR function.

Output:

You'll see the loss decrease as the network trains, and the final predictions should be close to `[0, 1, 1, 0]`, matching the XOR logic.

In backpropagation, we multiply by the derivative of the sigmoid function because of the **chain rule** in calculus, which allows us to compute how a small change in the weights affects the loss.

Let me break it down step by step to clarify why the derivative of the activation function (in this case, sigmoid) is necessary in the weight update process.

1. The Chain Rule in Backpropagation

When you're training a neural network, you're trying to adjust the weights to minimize the loss function. To do that, you need to calculate the gradient (or slope) of the loss function with respect to the weights. However, the weights don't directly affect the loss function; they affect the loss **indirectly** through the neurons and their activations.

In mathematical terms, for the output layer, the loss L depends on the weights $W^{(2)}$ through the activation $a^{(2)}$, which in turn depends on the input $z^{(2)}$ (a linear combination of weights and inputs). To calculate how a change in the weights $W^{(2)}$ affects the loss, we apply the **chain rule**:

$$\frac{\partial L}{\partial W^{(2)}} = \frac{\partial L}{\partial a^{(2)}} \cdot \frac{\partial a^{(2)}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial W^{(2)}}$$

Here's what each term represents:

- $\frac{\partial L}{\partial a^{(2)}}$: The gradient of the loss with respect to the output $a^{(2)}$. This is easy to compute as $a^{(2)} - y$ (the difference between the predicted value and the actual value).
- $\frac{\partial a^{(2)}}{\partial z^{(2)}}$: The gradient of the activation function (sigmoid) with respect to its input $z^{(2)}$.
- $\frac{\partial z^{(2)}}{\partial W^{(2)}}$: The gradient of the weighted sum with respect to the weights $W^{(2)}$.

2. **Why Multiply by the Sigmoid Derivative?**

The **sigmoid function** maps the linear combination $z^{(2)} = W^{(2)} \cdot a^{(1)} + b^{(2)}$ into the output $a^{(2)}$. The change in the loss with respect to the weights depends not only on the difference between the predicted and actual values, but also on how "sensitive" the sigmoid function is to changes in the input $z^{(2)}$.

The sensitivity of the sigmoid function is captured by its **derivative**:

$$\sigma'(z^{(2)}) = \sigma(z^{(2)}) \cdot (1 - \sigma(z^{(2)}))$$

Multiplying by this derivative helps us adjust the weights appropriately depending on how far we are from the correct output and how steep the sigmoid function is in that region.

For example:

- If $z^{(2)}$ is very large (far from the decision boundary), the sigmoid's derivative is small, and thus the weight changes will be small. This makes sense because the sigmoid is almost "saturated" in those regions, and further changes won't have much effect.
- If $z^{(2)}$ is near 0 (near the decision boundary), the sigmoid's derivative is large, so the weight changes will be larger because small adjustments near the decision boundary can significantly change the output.

3. **Final Formula for Output Layer Gradient**

So, when you calculate the error for the output layer, you compute:

$$\delta^{(2)} = \frac{\partial L}{\partial z^{(2)}} = (a^{(2)} - y) \cdot \sigma'(z^{(2)})$$

Where:

- $(a^{(2)} - y)$ is the error based on the difference between the predicted and actual values.
- $\sigma'(z^{(2)})$ adjusts the magnitude of the error according to the steepness of the sigmoid function at that point.

4. **Intuition Behind This**

- The derivative of the sigmoid acts as a **scaling factor**. If the output neuron is confident (e.g., output near 1 or 0), its derivative is small, meaning weight updates should be cautious.
- If the output is unsure (e.g., output around 0.5), its derivative is large, meaning more aggressive weight updates are necessary to learn more quickly.

Conclusion

Multiplying by the derivative of the sigmoid ensures that we adjust the weights based not only on the error but also on how the activation function (sigmoid) behaves. This leads to more efficient learning and helps the network converge during training.

Let me know if you'd like further clarifications or another example!

This equation represents the **backpropagation** step for hidden layers in a neural network. It shows how to compute the error term $\delta^{(1)}$ for the hidden layer, based on the error term $\delta^{(2)}$ from the output layer. Let's go step by step to understand how this formula is derived mathematically.

Recap of Variables and Terms:

- $\delta^{(1)}$: The error term for the neurons in the hidden layer.
- $\delta^{(2)}$: The error term for the neurons in the output layer.
- $W^{(2)}$: The weights connecting the hidden layer to the output layer.
- $z^{(1)}$: The linear combination (weighted input) for the neurons in the hidden layer.
- $\sigma'(z^{(1)})$: The derivative of the activation function (e.g., sigmoid) applied to the hidden layer, with respect to the input $z^{(1)}$.

Step-by-Step Derivation of $\delta^{(1)}$

1. Backpropagation Through Layers

In backpropagation, the goal is to compute how much each neuron in the hidden layer contributes to the error in the output.

We start by calculating the error in the output layer ($\delta^{(2)}$) and propagate it backward through the network to adjust the weights and update the neurons in the hidden layer ($\delta^{(1)}$).

2. Chain Rule and Layer Dependency

We use the **chain rule** from calculus to propagate the error backward. The idea is to calculate how the loss function L changes with respect to the weighted sum $z^{(1)}$ of the hidden layer neurons.

We know the following relationships:

- $z^{(2)} = W^{(2)}a^{(1)} + b^{(2)}$, where $a^{(1)}$ is the activation from the hidden layer.
- The loss L depends on the activations $a^{(2)}$ of the output layer, which depends on the weighted inputs $z^{(2)}$, which are in turn affected by the activations of the hidden layer $a^{(1)}$.

So, we need to compute the derivative of the loss L with respect to the hidden layer's weighted input $z^{(1)}$.

$$\delta^{(1)} = \frac{\partial L}{\partial z^{(1)}} = \frac{\partial L}{\partial a^{(2)}} \cdot \frac{\partial a^{(2)}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial a^{(1)}} \cdot \frac{\partial a^{(1)}}{\partial z^{(1)}}$$

3. Propagating Error to the Hidden Layer:

The error term $\delta^{(1)}$ depends on how much the hidden layer's activation $a^{(1)}$ contributes to the error in the output layer. Here's how this works:

1. **Output Layer Error**: The error term for the output layer is $\delta^{(2)} = \frac{\partial L}{\partial z^{(2)}}$. We've already computed this error for the output layer during backpropagation from the loss.
2. **How Hidden Layer Activations Affect the Output**: The activations of the hidden layer $a^{(1)}$ affect the output layer through the weights $W^{(2)}$. To compute how much each hidden layer neuron affects the output error, we take the gradient of $z^{(2)}$ with respect to $a^{(1)}$:

   $$\frac{\partial z^{(2)}}{\partial a^{(1)}} = W^{(2)}$$

   This tells us how the output error $\delta^{(2)}$ propagates back through the weights $W^{(2)}$ to the hidden layer. Specifically, we multiply the output layer error $\delta^{(2)}$ by the transpose of $W^{(2)}$:

   $$(W^{(2)})^T \delta^{(2)}$$

   This term shows how the error is distributed to each hidden layer neuron based on the weights connecting the hidden layer to the output.

3. **Derivative of the Activation Function**: After propagating the error through the weights, we need to account for how the hidden layer neurons respond to this error. Since the hidden layer neurons use an activation function (like sigmoid), we need to multiply by the derivative of the activation function with respect to the input $z^{(1)}$:

$$\frac{\partial a^{(1)}}{\partial z^{(1)}} = \sigma'(z^{(1)})$$

   The derivative $\sigma'(z^{(1)})$ tells us how sensitive each neuron in the hidden layer is to the propagated error. So, we scale the propagated error by this derivative.

4. Final Formula for Hidden Layer Error

Putting everything together, the error term for the hidden layer neurons becomes:

$$\delta^{(1)} = (W^{(2)})^T \delta^{(2)} \cdot \sigma'(z^{(1)})$$

Where:

- $(W^{(2)})^T \delta^{(2)}$ is the error propagated back from the output layer through the weights.
- $\sigma'(z^{(1)})$ scales this error according to the sensitivity of the hidden layer neurons.

Summary of the Formula

The formula $\delta^{(1)} = (W^{(2)})^T \delta^{(2)} \cdot \sigma'(z^{(1)})$ describes how the error in the hidden layer ($\delta^{(1)}$) is calculated during backpropagation:

- First, the error from the output layer ($\delta^{(2)}$) is propagated backward through the weights $W^{(2)}$.
- Then, the propagated error is scaled by the derivative of the activation function applied to the hidden layer's weighted input $z^{(1)}$, which determines how sensitive each hidden neuron is to the propagated error.

This step-by-step process ensures that we adjust the weights in the hidden layer based on how much each hidden neuron contributes to the output error.

Let me know if you'd like further clarification!

Good question! Let's clarify why the derivative of the sigmoid function, $\sigma(x) \cdot (1 - \sigma(x))$, is often written as $x \cdot (1 - x)$ in code.

The Sigmoid Function

The sigmoid function is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Its derivative is:

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$$

This derivative tells us how the output of the sigmoid changes with respect to the input x.

Why Do We Use $x \cdot (1 - x)$ in Code?

In the code implementation, when you see $x \cdot (1 - x)$, **x is not the input x to the sigmoid function**; it is actually the **output of the sigmoid function**.

Let's break this down:

1. When you compute the derivative of the sigmoid during backpropagation, you're not using the raw input x. Instead, you're using the **output** of the sigmoid function, which is $\sigma(x)$. So, in code, x actually refers to the sigmoid output $\sigma(x)$.

2. Once the sigmoid is applied to x, the result is $\sigma(x)$. Therefore, when calculating the derivative, you can simply write:

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$$

If we already know that $\sigma(x) = x$ (since x here represents the sigmoid output), the derivative becomes:

$$\sigma'(x) = x \cdot (1 - x)$$

So, in code, instead of recalculating $\sigma(x)$, we use the existing output value, which simplifies the expression to $x \cdot (1 - x)$.

Example code

```
import numpy as np

# Sigmoid function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

```
# Derivative of sigmoid
def sigmoid_derivative(output):
    return output * (1 - output)

# Example usage
x = np.array([0.5])  # Input
output = sigmoid(x)  # Apply sigmoid
deriv = sigmoid_derivative(output)  # Compute derivative using the output

print("Sigmoid output:", output)
print("Sigmoid derivative:", deriv)
```

Key Points:

- In the code, x · (1−x)x \cdot (1 - x)x · (1−x) refers to the derivative of the sigmoid, where xxx is the **output** of the sigmoid, not the input.
- The formula σ(x) · (1−σ(x))\sigma(x) \cdot (1 - \sigma(x))σ(x) · (1−σ(x)) is the mathematical derivative, but in practice, you already have σ(x)\sigma(x)σ(x) from the forward pass, so you use x · (1−x)x \cdot (1 - x)x · (1−x) in the backpropagation step.

Let me know if you need further clarification!